廣東工業大學

实 验 报 告

课程名称 操作系统课程设计

学生学院 计算机学院

专业班级 人工智能

学　　号 3121005358

学生姓名 欧炜标

指导教师 张伟文

2023 年 6 月 10 日

目录

实验一　进程调度（时间片和动态优先）

一、实验目的

编写并调试一个模拟的进程调度程序，以加深对进程的概念及进程调度算法的理解.

二、实验内容

1. 调试运行“时间片轮转”调度算法，给出运行结果。
2. 采用“时间片轮转”调度算法对进程进行调度。每个进程有一个进程控制块（ PCB）表示。进程控制块可以包含如下信息：进程名、到达时间、需要运行时间、已用 CPU 时间、进程状态等等。
3. 每个进程的状态可以是就绪 W(Wait)、运行 R(Run)、或完成 F(Finish)三种状态之一。 显示进程运行过程，以及进程的带权周转时间和系统的平均带权周转时间。

三、实现思路

时间片轮转算法是轮流为进程服务的，当就绪队列的进程 A 的时间片用完，将就绪队列中 A 进程后面的进程取出来运行，并且将进程 A 插入到就绪队列的末尾。但此时如果有新的进程到达，将新进程插入到当前的就绪队列的末尾。由于本程序在程序开始时就已经输入了所有要运行的进程的信息，考虑到进程的到达时间不同，于是创建了两条链表，一条是包含所有进程信息的链表 all_process，一条是就绪队列链表 ready。

运行过程如下：

1. 在每次运行前，检查 all_process 中是否有进程的到达时间与当前时间相符，如果有，则将其插入到 ready 的队尾；否则进行第二步。
2. 取 ready 队首进程 p，并将其运行。
3. 当时间片用完后，如果 p 已经达到 cpu 运行时间，则将 p 已运行完毕，将 p 销毁，否则将 p 插入到 ready 的末尾。
4. 运行第一步，直到 all_process 和 ready 链表为空。

四、主要的数据结构

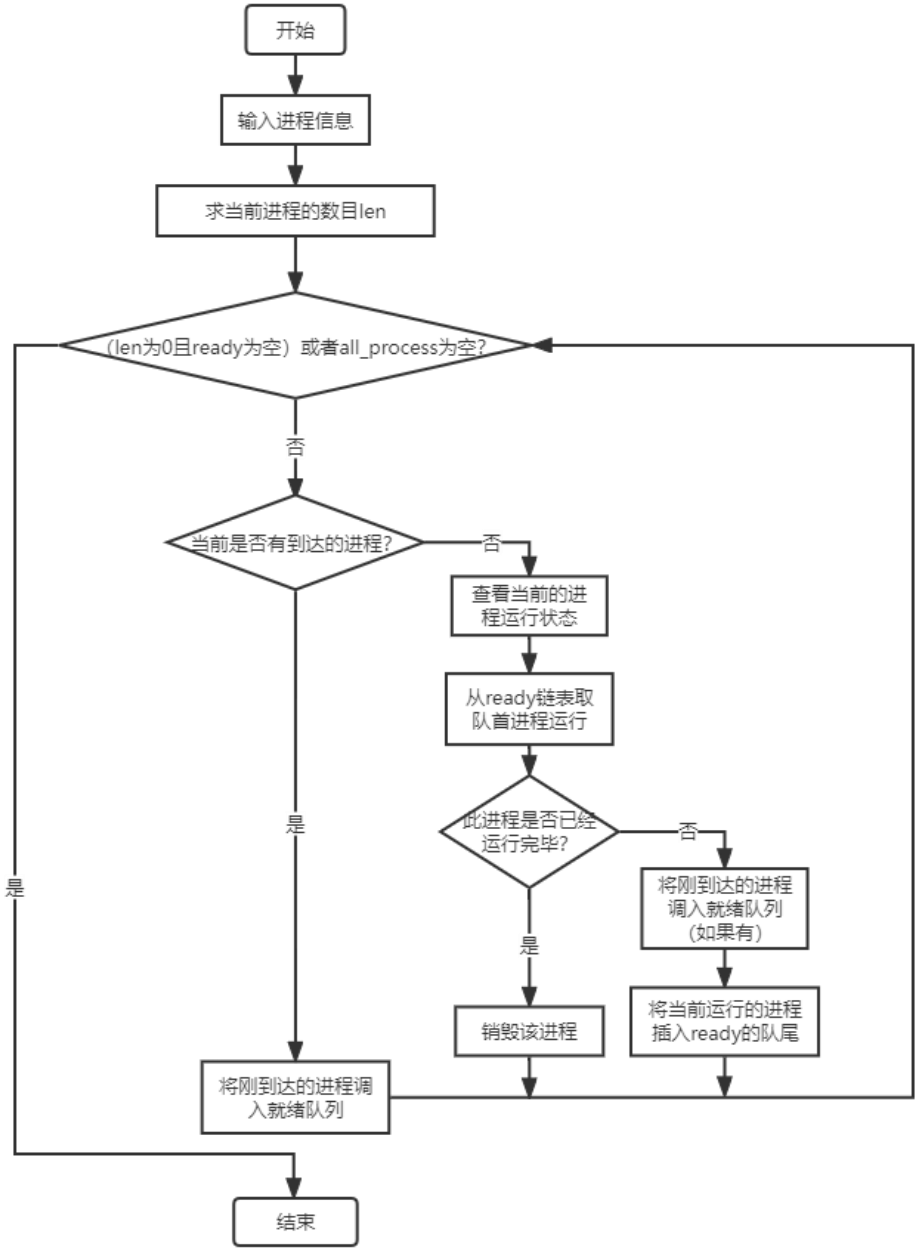
本算法用到的结构体定义如下：

```
struct pcb { /*  定义进程控制块 PCB */
    char name[10];
    char state;         //  状态
    int arrtime;        //  到达时间
    int ntime;          //  服务时间
    int rtime;          //  已用 CPU 时间
    int ftime;          //  完成时间，用于计算周转时间
    struct pcb* link;
}
```

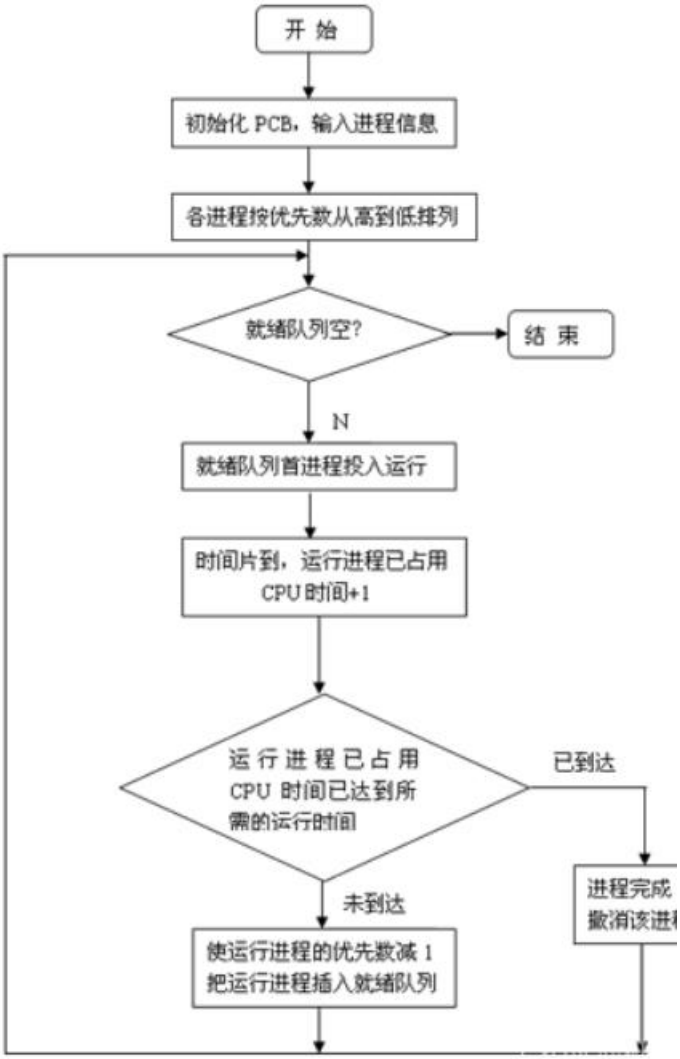
注：因时间片轮转具体算法代码在实验手册上已经明确给出，在此不再赘述。

五、算法流程图

时间片算法：

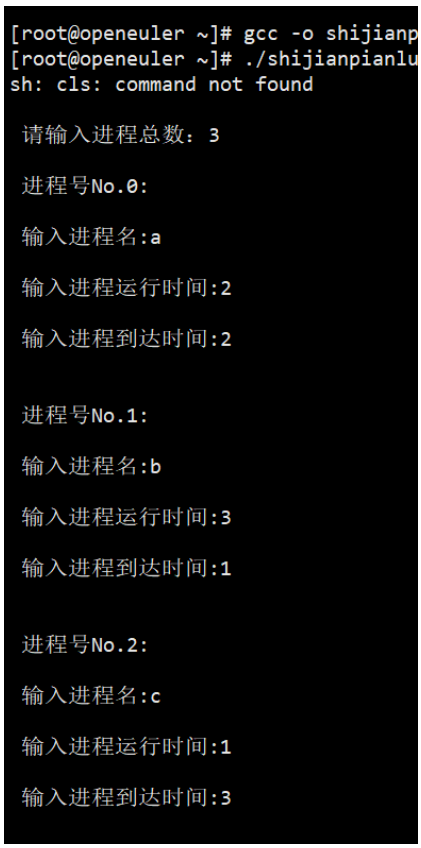

开始
输入进程信息
求当前进程的数目len
(len为0且ready为空）或者all_process为空?
否
当前是否有到达的进程?
否
查看当前的进程运行状态
从ready链表取队首进程运行
此进程是否已经运行完毕?
否
将刚到达的进程调入就绪队列(如果有)
是
销毁该进程
将当前运行的进程插入ready的队尾
是
将刚到达的进程调入就绪队列
是
结束

动态优先数算法：

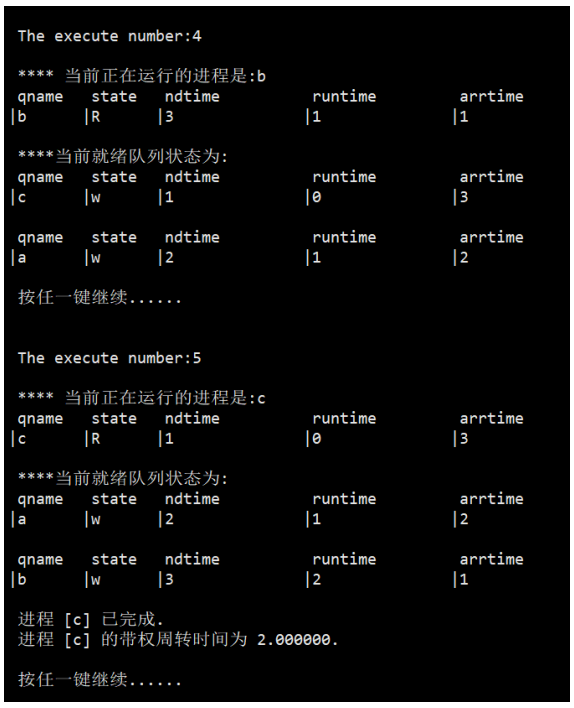


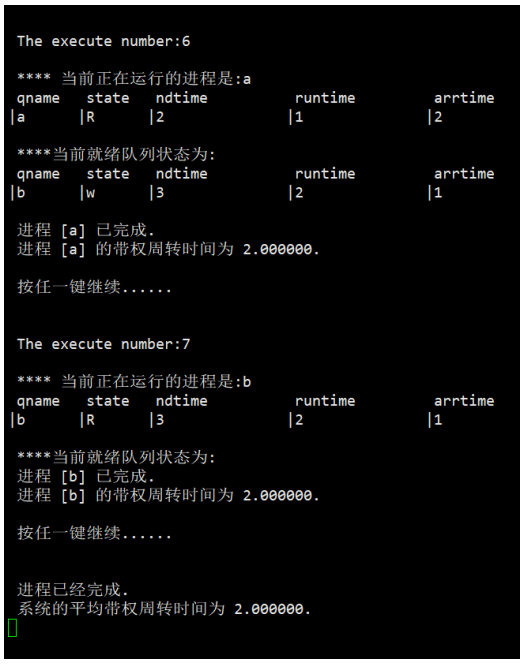
六、运行与测试

时间片轮转算法：

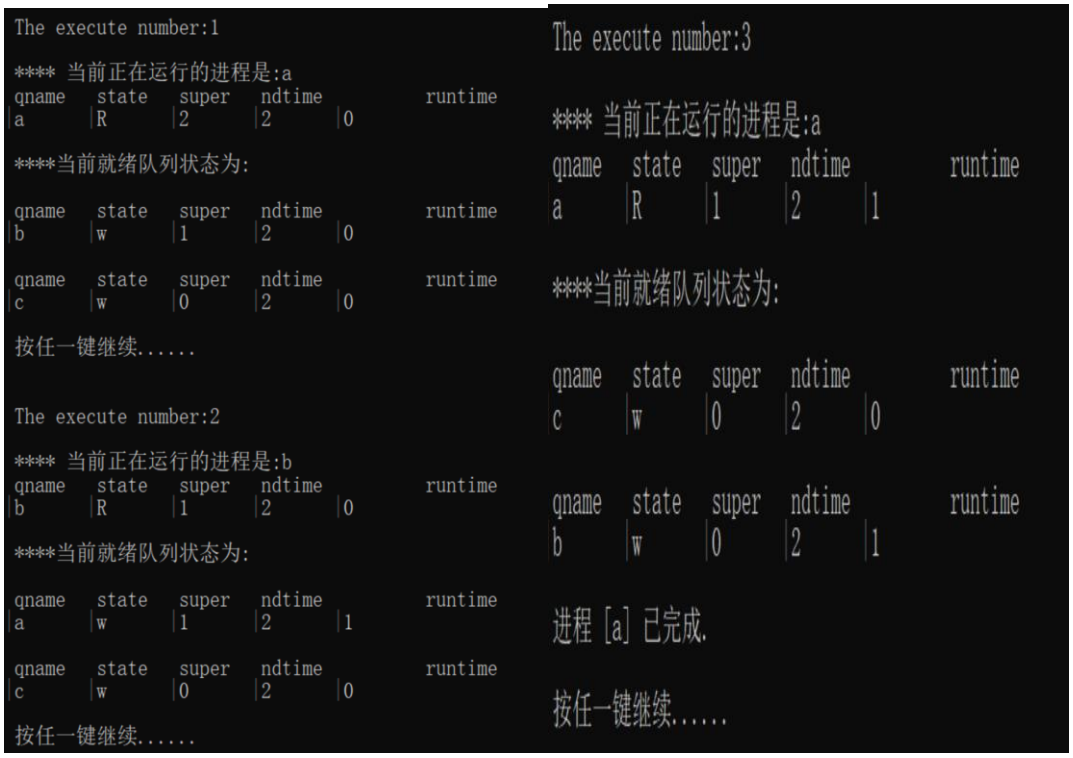

[root@openeuler ~]# gcc -o shijianp
[root@openeuler ~]# ./shijianpianlu
sh: cls: command not found
请输入进程总数：3
进程号No.0:
输入进程名:a
输入进程运行时间:2
输入进程到达时间:2
进程号No.1:
输入进程名:b
输入进程运行时间:3
输入进程到达时间:1
进程号No.2:
输入进程名:c
输入进程运行时间:1
输入进程到达时间:3


进程号No.2:
输入进程名:c
输入进程运行时间:1
输入进程到达时间:3
The execute number:1
The execute number:2
**** 当前正在运行的进程是:b
qname state ndtime runtime arrtime
|b |R |3 |0 |1
****当前就绪队列状态为:
按任一键继续.....
The execute number:3
**** 当前正在运行的进程是:a
qname state ndtime runtime arrtime
|a |R |2 |0 |2
****当前就绪队列状态为:
qname state ndtime runtime arrtime
|b |w |3 |1 |1
按任一键继续.....


The execute number:4
**** 当前正在运行的进程是:b
qname state ndtime runtime arrtime
|b |R |3 |1 |1
****当前就绪队列状态为:
qname state ndtime runtime arrtime
|c |w |1 |0 |3
qname state ndtime runtime arrtime
|a |w |2 |1 |2
按任一键继续.....
The execute number:5
**** 当前正在运行的进程是:c
qname state ndtime runtime arrtime
|c |R |1 |0 |3
****当前就绪队列状态为:
qname state ndtime runtime arrtime
|a |w |2 |1 |2
qname state ndtime runtime arrtime
|b |w |3 |2 |1
进程 [c] 已完成.
进程 [c] 的带权周转时间为 2.000000.
按任一键继续.....


The execute number:6
**** 当前正在运行的进程是:a
qname state ndtime runtime arrtime
|a |R |2 |1 |2
****当前就绪队列状态为:
qname state ndtime runtime arrtime
|b |w |3 |2 |1
进程 [a] 已完成.
进程 [a] 的带权周转时间为 2.000000.
按任一键继续.....
The execute number:7
**** 当前正在运行的进程是:b
qname state ndtime runtime arrtime
|b |R |3 |2 |1
****当前就绪队列状态为:
进程 [b] 已完成.
进程 [b] 的带权周转时间为 2.000000.
按任一键继续.....
进程已经完成.
系统的平均带权周转时间为 2.000000.

动态优先数算法：



七、总结

通过这次实验，加深了我对时间片轮转算法的认识。

实验二　动态分区分配方式的模拟

一、实验目的

了解动态分区分配方式中的数据结构和分配算法，并进一步加深对动态分区存储管理方式及其实现过程的理解

二、实验内容

1. 用 C 语言分别实现采用首次适应算法和最佳适应算法的动态分区分配过程和回收过程。其中，空闲分区通过空闲分区链（表）来管理；在进行内存分配时，系统优先使用空闲区低端的空间。
2. 假设初始状态下，可用的内存空间为 640KB，并有下列的请求序列：
   - 作业 1 申请 130KB

- 作业 2 申请 60KB
- 作业 3 申请 100KB
- 作业 2 释放 60KB
- 作业 4 申请 200KB
- 作业 3 释放 100KB
- 作业 1 释放 130KB
- 作业 5 申请 140KB
- 作业 6 申请 60KB
- 作业 7 申请 50KB
- 作业 8 申请 60KB

请分别采用首次适应算法和最佳适应算法进行内存的分配和回收，要求每次分配和回收后显示出空闲内存分区链的情况。

三、实现思路

为了方便管理，我们需要为每一个分区创建一个控制块，用来保存当前分区的信息，比如分区大小，是否已经被分配等等。再将这些分区连接起来，形成分区链，方便管理。管理分区的控制块的数据结构见下文。

FF（First Fit 首次适应算法）要求空闲分区链以地址递增的次序链接，在分配内存时，从链首开始顺序查找，直至找到一个大小能满足要求的空闲分区为止。在进行分区时，需要创建一个分区控制块节点，用来存储当前分区的信息，并且修改空闲分区的相关信息。

BF（Best Fit 最佳适应算法）与 FF 的不同之处在于其进行分区时是根据有序的空闲分区链进行分配的。所谓有序的空闲分区链是指将当前所有的空闲分区按其空闲大小从小到大的顺序形成一空闲分区链。

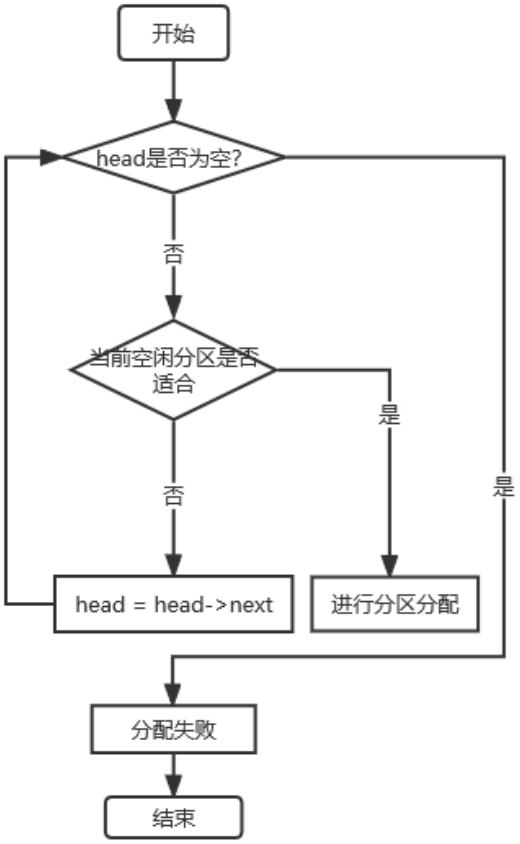
四、主要的数据结构

```
struct area
{
    int id;                 // 编号
    int addr_front;        //首地址
    int size;               //分区大小
    int flag;               //分配标志
    char* name;
    struct area* front; //上一分区
```

```
    struct area* next;   //下一分区
};
```

五、算法流程图

分区流程图：（假设 head 指向当前的空闲分区的头节点）



FF 与 BF 的不同之处在于查找空闲分区的方法不同，FF 的空闲分区链是自然连接的，而 BF 的空闲分区链是按照大小来进行排序连接的，在形成空闲分区链时注意两者的构建方法不同。但两者分区的大概流程是相同的。

六、运行与测试

注：

- 因华为云的代金券已用完，无法在华为云 openEuler 上完成实验，因此本次实验环境为 Windows，所用软件为 visual studio 2022。

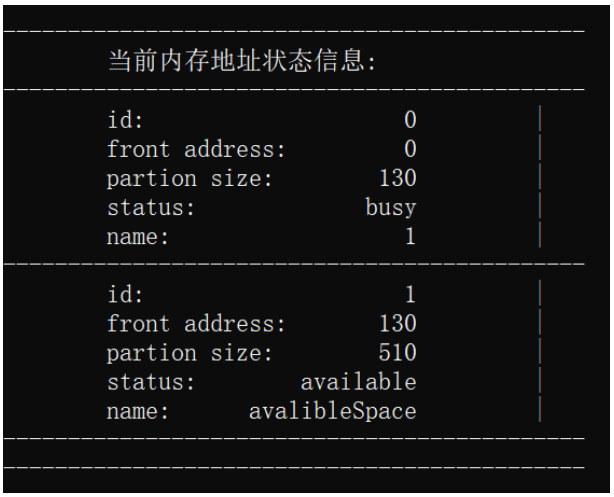
6.1 First Fit 首次适应算法

刚启动程序，内存地址信息如下：

当前的内存空间为 640KB（实验中省略了内存单位，默认为 KB）；起始地址为 0.按照实验书的要求依次为其分配和回收内存的过程如下。

作业 1 申请 130KB：

根据 FF(First Fit 算法的缩写，写文用此缩写代指首次适应算法)算法的分配过程，原来的 640KB 空间被划分为两个分区，一个区的大小为 130KB，分配给作业 1，另一个则处于空闲状态。剩余 510KB。



作业 2 申请 60KB：

申请的空间 60KB < 可用的空间 510KB，进行内存分配。剩余 450KB。

```
-----------------------------------------
       当前内存地址状态信息:
-----------------------------------------
       id:                    0          |
       front address:         0          |
       partion size:        130          |
       status:             busy          |
       name:                  1          |
-----------------------------------------
       id:                    1          |
       front address:       130          |
       partion size:         60          |
       status:             busy          |
       name:                  2          |
-----------------------------------------
       id:                    2          |
       front address:       190          |
       partion size:        450          |
       status:        available          |
       name:      availibleSpace         |
-----------------------------------------
-----------------------------------------
```

作业 3 申请 100KB：

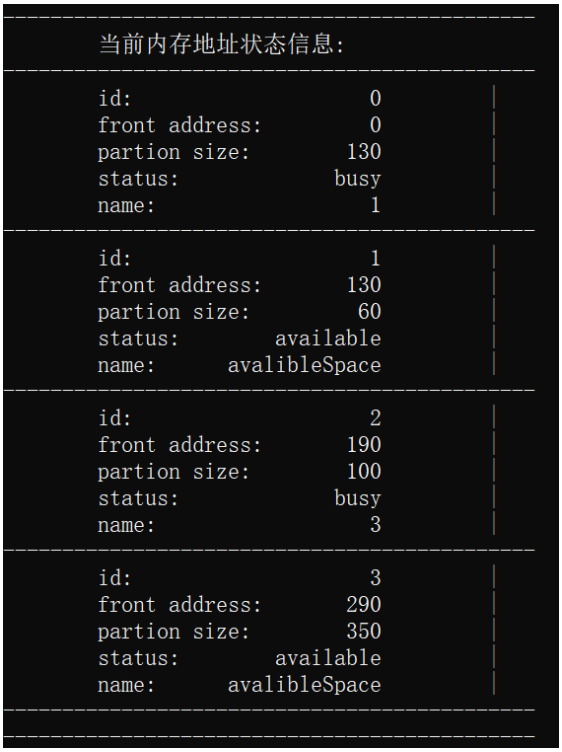
申请的空间 100KB < 可用的空间 450KB，进行内存分配。剩余 350KB。

```
-----------------------------------------
       当前内存地址状态信息:
-----------------------------------------
       id:                    0          |
       front address:         0          |
       partion size:        130          |
       status:             busy          |
       name:                  1          |
-----------------------------------------
       id:                    1          |
       front address:       130          |
       partion size:         60          |
       status:             busy          |
       name:                  2          |
-----------------------------------------
       id:                    2          |
       front address:       190          |
       partion size:        100          |
       status:             busy          |
       name:                  3          |
-----------------------------------------
       id:                    3          |
       front address:       290          |
       partion size:        350          |
       status:        available          |
       name:      availibleSpace         |
-----------------------------------------
-----------------------------------------
```

作业 2 释放 60KB：

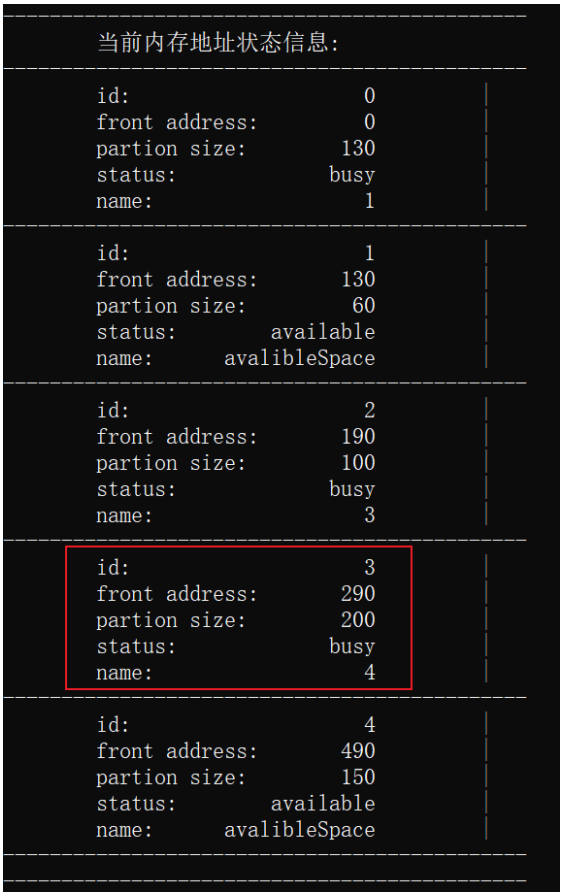
由下图可知，原本作业 2 的内存空间（id 号为 1）已经被释放掉了，当前的状态（status）为可用地址空间（availibleSpace）。当前有两个可用的分区，空间

大小分别为 60，350.



作业 4 申请 200KB：

新分配的内存信息如下图中红色框的部分。由于 200 < 350，因此从第二个可用的分区分配内存。当前有两个可用的分区，空间大小分别为 60，150.



作业 3 释放 100KB：

释放后的内存空间如下图红色框所示。由于在释放作业 3 前，已有空闲的地址，id 号为 1，起始地址为 130，终止地址为 189。而作业 3 的起始地址为 190，释放完作业 3 后，id 号为 1 的和 id 号为 2 的地址空间是连续的，于是合并为一个可用的地址空间，大小为 160。当前可用地址有两个分区，大小分别为 160，150。

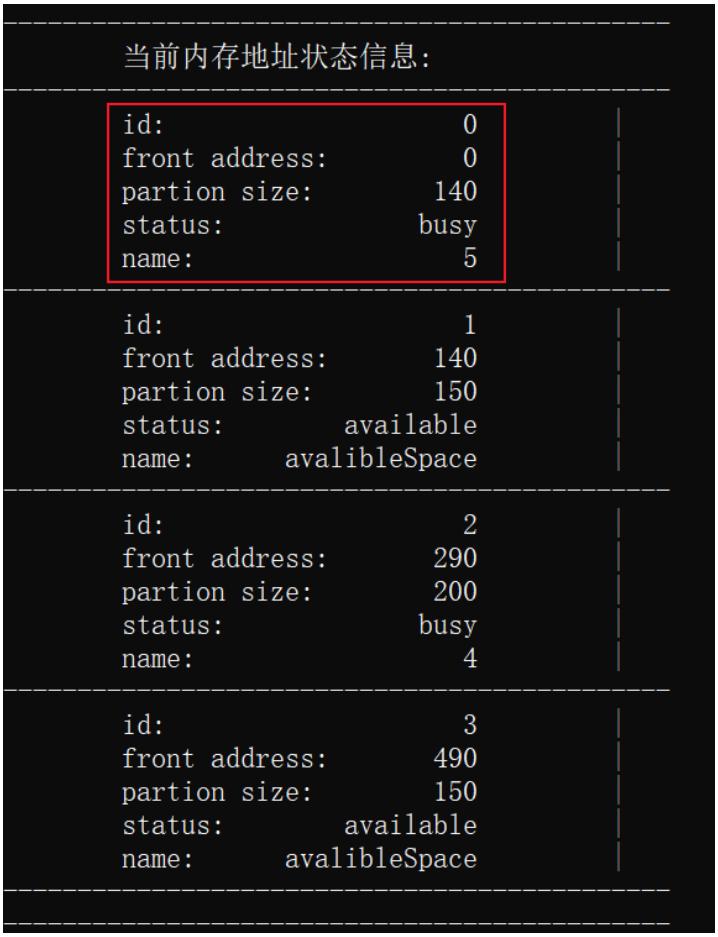
```
-----------------------------------------
       当前内存地址状态信息:
-----------------------------------------
       id:                    0
       front address:         0
       partion size:        130
       status:             busy
       name:                  1
-----------------------------------------
       id:                    1
       front address:       130
       partion size:        160
       status:        available
       name:      availibleSpace
-----------------------------------------
       id:                    2
       front address:       290
       partion size:        200
       status:             busy
       name:                  4
-----------------------------------------
       id:                    3
       front address:       490
       partion size:        150
       status:        available
       name:      availibleSpace
-----------------------------------------
-----------------------------------------
```

作业 1 释放 130KB：

```
-----------------------------------------
       当前内存地址状态信息:
-----------------------------------------
       id:                    0
       front address:         0
       partion size:        290
       status:        available
       name:      availibleSpace
-----------------------------------------
       id:                    1
       front address:       290
       partion size:        200
       status:             busy
       name:                  4
-----------------------------------------
       id:                    2
       front address:       490
       partion size:        150
       status:        available
       name:      availibleSpace
-----------------------------------------
-----------------------------------------
```

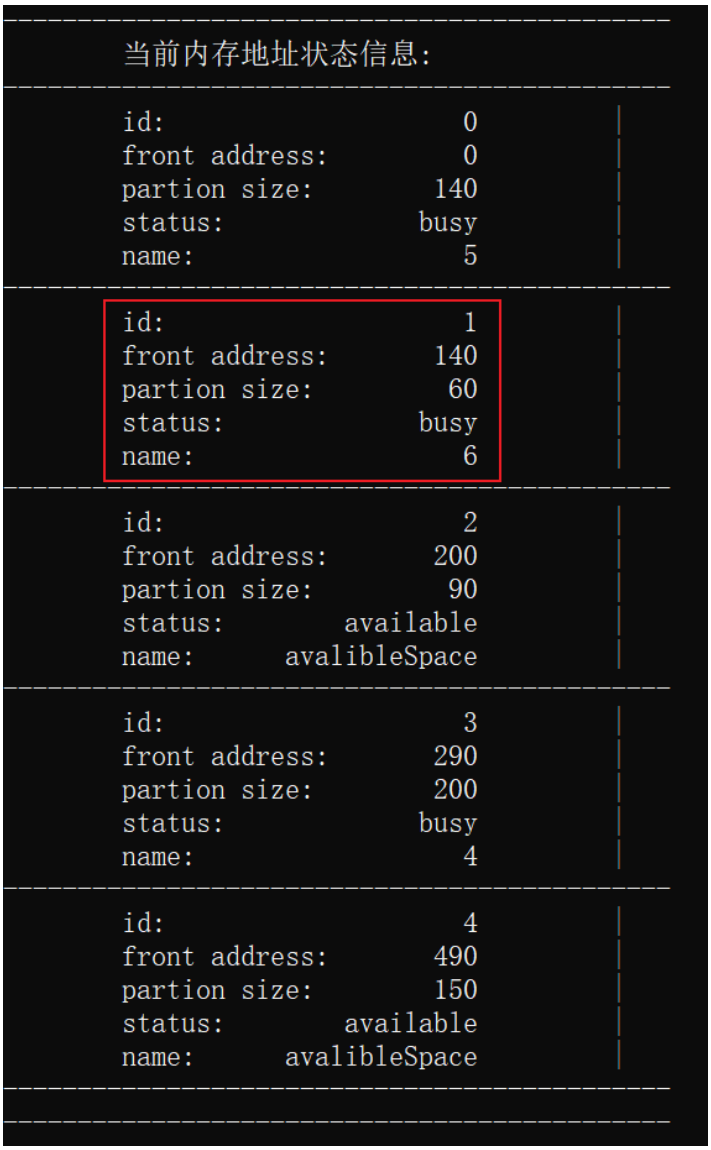
释放后的内存空间如上图红色框所示。由于在释放作业 1 前，已有空闲的地址，id 号为 1，起始地址为 130，终止地址为 289。而作业 1 的起始地址为 0，终止地址为 129。释放完作业 1 后，id 号为 0 的和 id 号为 1 的地址空间是连续的，于是合并为一个可用的地址空间，大小为 290。当前可用地址有两个分区，大小分别为 290，150.

作业 5 申请 140KB：



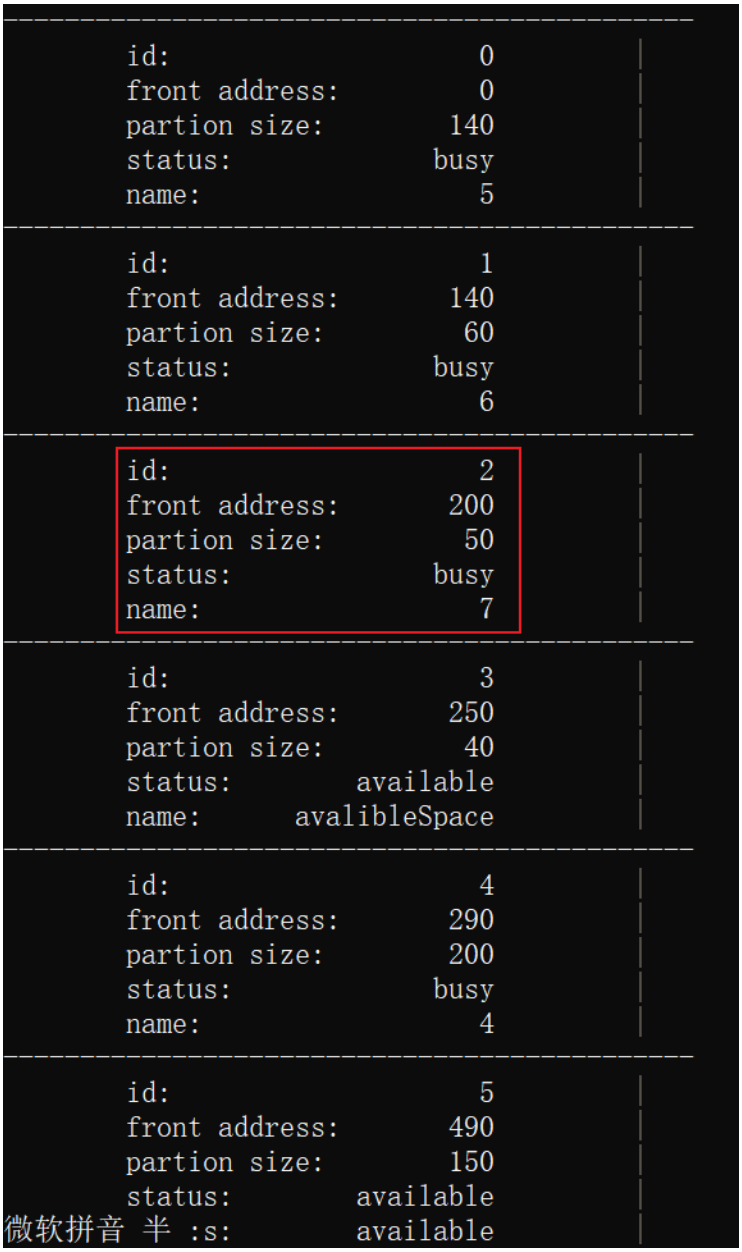
分配的情况如上图红色框所示。根据 FF 算法，从第一个空闲分区开始顺序查找，直到找到一个大小能满足要求的空闲分区为止。第一个空闲分区大小为 290，第二个空闲分区大小为 150，均满足要求，于是选择第一个空闲分区进行分配。当前剩余两个空闲分区，大小分别为 150，150.

作业 6 申请 60KB



当前的两个空闲分区均满足要求，根据 FF 算法，选择第一个空闲分区，分配情况如上图所示。剩余两个空闲分区，大小分别为 90，150.

作业 7 申请 50KB



注：因信息过长，无法完全截图，因此保留关键的截图信息。

当前的两个空闲分区均满足要求，根据 FF 算法，选择第一个空闲分区，分配情况如上图所示。剩余两个空闲分区，大小分别为 40，150.

作业 8 申请 60KB：

```
front address:       200
partion size:         50
status:             busy
name:                 7
-------------------------------------
id:                   3
front address:       250
partion size:         40
status:        available
name:      avalibleSpace
-------------------------------------
id:                   4
front address:       290
partion size:        200
status:             busy
name:                 4
-------------------------------------
id:                   5
front address:       490
partion size:         60
status:             busy
name:                 8
-------------------------------------
id:                   6
front address:       550
partion size:         90
status:        available
name:      avalibleSpace
-------------------------------------
```

当前有两个分区，大小分别为 40，150。而要求的内存为 60，第一个空闲分区不满足要求，因此选择第二个。分配后剩余两个空闲分区，大小分别为 40，90。

6.2 Best Fit 最佳适应算法

Best Fit 算法（一下简称 BF）要求将所有的空闲分区按其容量以从小到大的顺序形成一空闲分区链，再进行顺序查找。

作业 1 申请 130KB：

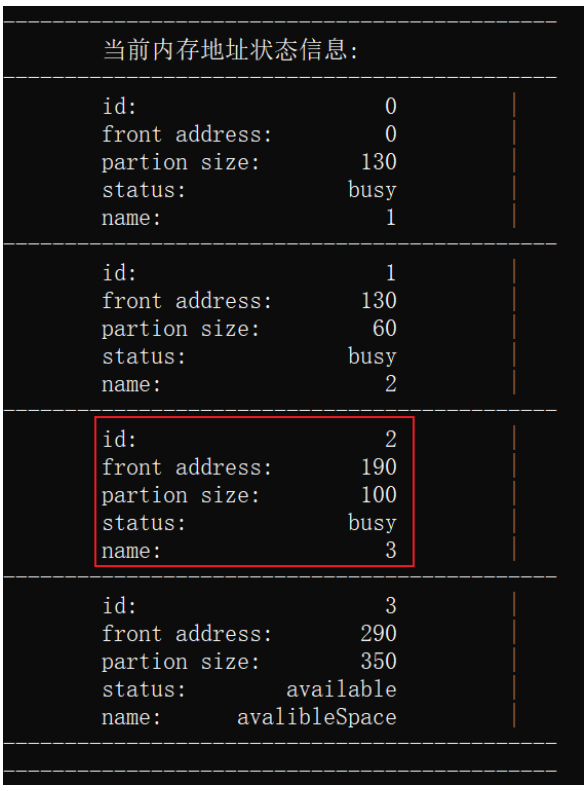
```
-----------------------------------------
       当前内存地址状态信息:
-----------------------------------------
       id:                   0
       front address:        0
       partion size:       130
       status:            busy
       name:                 1
-----------------------------------------
       id:                   1
       front address:      130
       partion size:       510
       status:       available
       name:     avalibleSpace
-----------------------------------------
-----------------------------------------
```

根据 BF 算法的分配过程，原来的 640KB 空间被划分为两个分区，一个区的大小为 130KB，分配给作业 1，另一个则处于空闲状态。分配后只有一个空闲分区，大小为 510KB。

作业 2 申请 60KB：

申请的空间 60KB < 可用的空间 510KB，进行内存分配。分配后只有一个空闲分区，大小为 450KB。

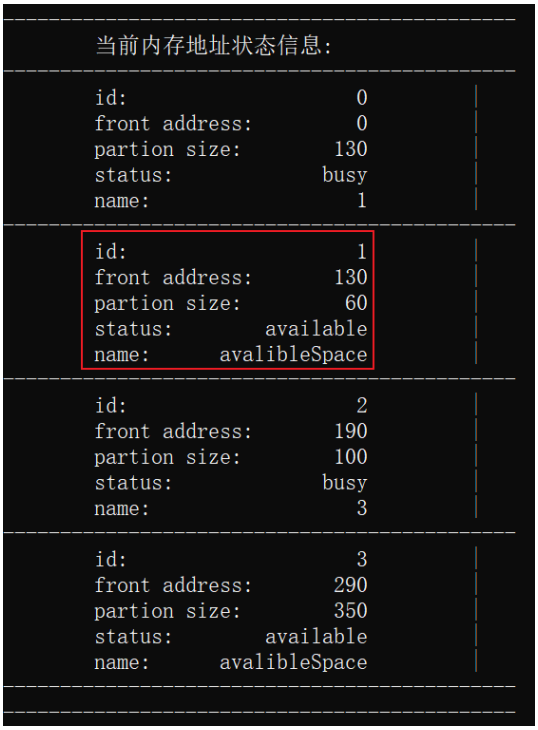
作业 3 申请 100KB：



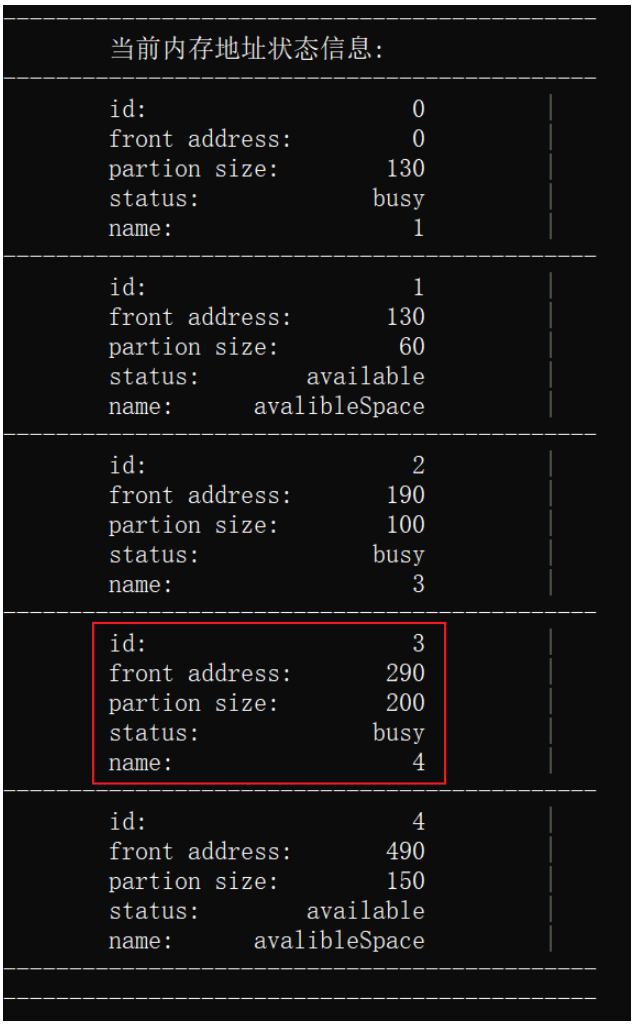
申请的空间 100KB < 可用的空间 450KB，进行内存分配。分配后只有一个空闲分区，大小为 350KB。

作业 2 释放 60KB：

由下图可知，原本作业 2 的内存空间（id 号为 1）已经被释放掉了，当前的状态（status）为可用地址空间（availibleSpace）。当前有两个可用的分区，空间大小分别为 60，350.

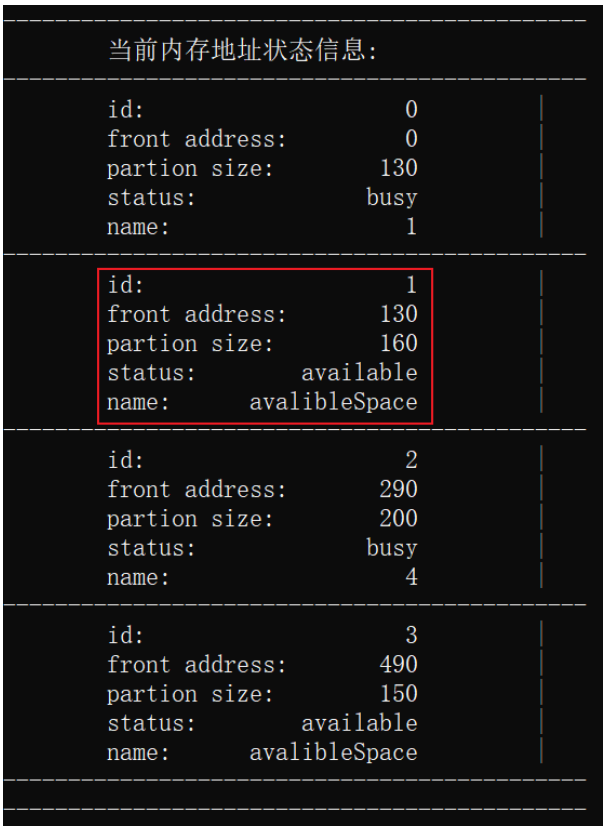


作业 4 申请 200KB：



新分配的内存信息如上图中红色框的部分。由于 200 < 350，因此从第二个可用的分区分配内存。当前有两个可用的分区，空间大小分别为 60，150.

作业 3 释放 100KB：

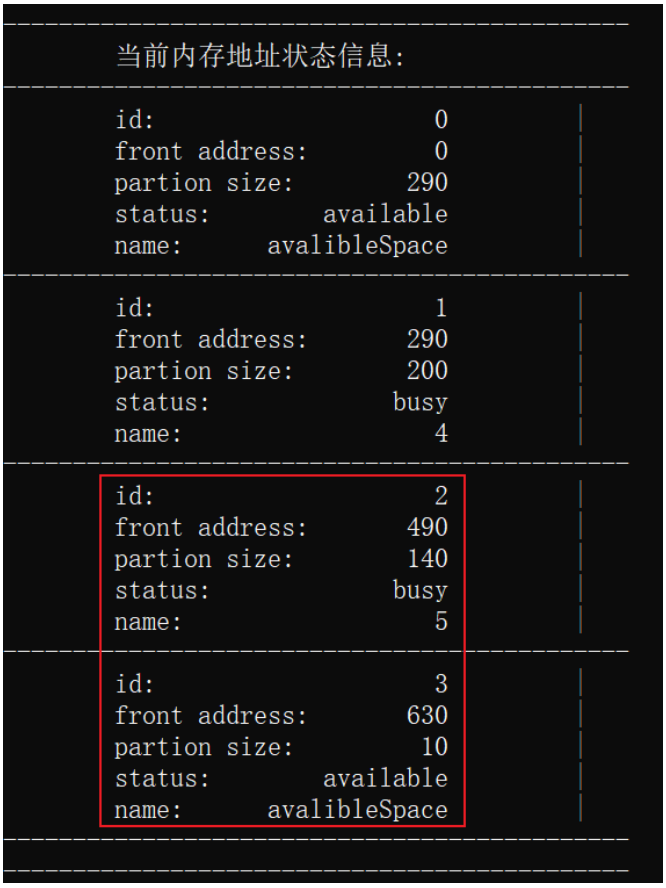


释放后的内存空间如上图红色框所示。由于在释放作业 3 前，已有空闲的地址，id 号为 1，起始地址为 130，终止地址为 189。而作业 3 的起始地址为 190，释放完作业 3 后，id 号为 1 的和 id 号为 2 的地址空间是连续的，于是合并为一个可用的地址空间，大小为 160。当前可用地址有两个分区，大小分别为 160，150.

作业 1 释放 130KB：

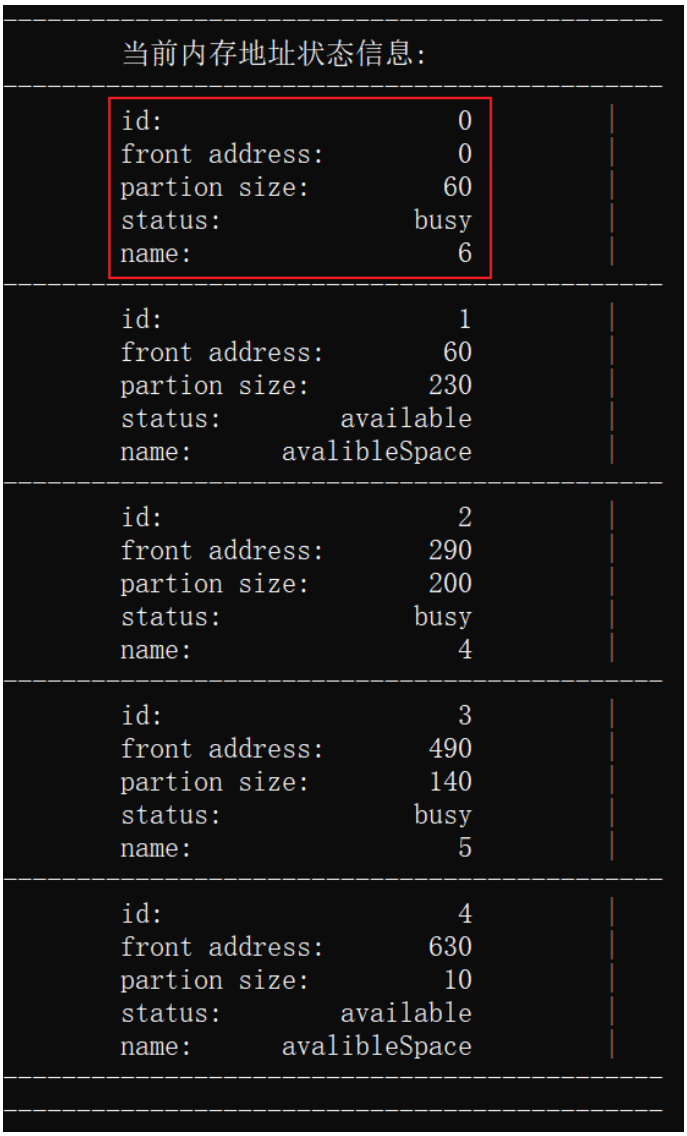
释放后的内存空间如上图红色框所示。由于在释放作业 1 前，已有空闲的地址，id 号为 1，起始地址为 130，终止地址为 289。而作业 1 的起始地址为 0，终止地址为 129。释放完作业 1 后，id 号为 0 的和 id 号为 1 的地址空间是连续的，于是合并为一个可用的地址空间，大小为 290。当前可用地址有两个分区，大小分别为 290，150.

作业 5 申请 140KB：



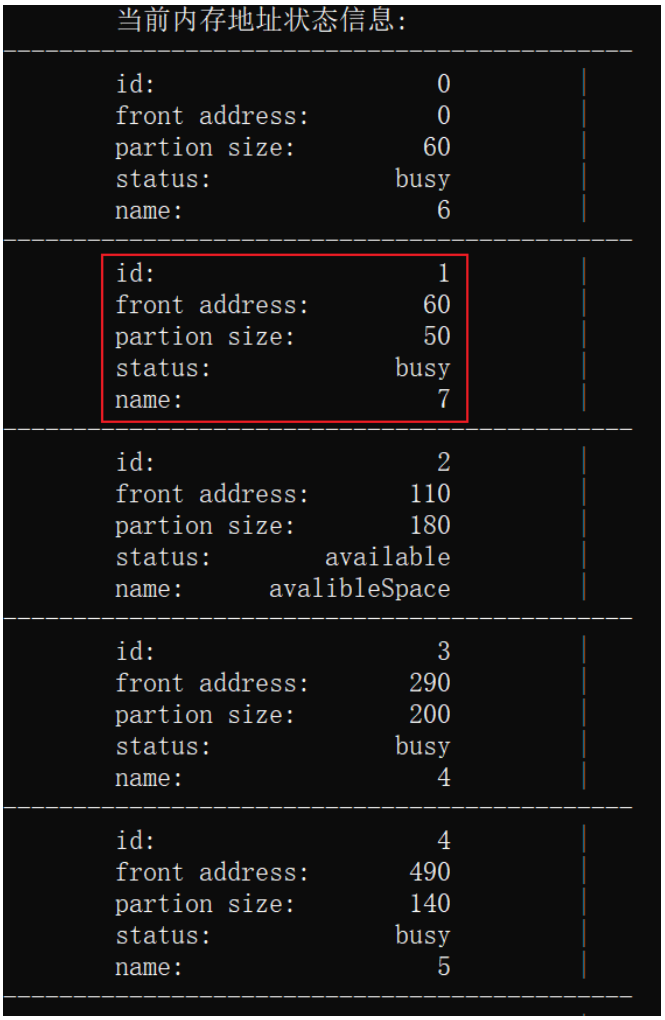
分配的情况如上图红色框所示。根据 BF 算法，将所有的空闲分区按其容量以从小到大的顺序形成一空闲分区链，再进行顺序查找。因此在空闲分区链中，第一个空闲分区大小为 150，第二个空闲分区大小为 290，均满足要求，于是选择第一个空闲分区进行分配。当前剩余两个空闲分区，大小分别为 10，290.

作业 6 申请 60KB



当前的两个空闲分区链中第一个分区大小为 10，不满足要求；第二个分区大小为 290，满足要求，于是对第二个空闲分区进行分配。分配情况如上图所示。剩余两个空闲分区，大小分别为 10，230.

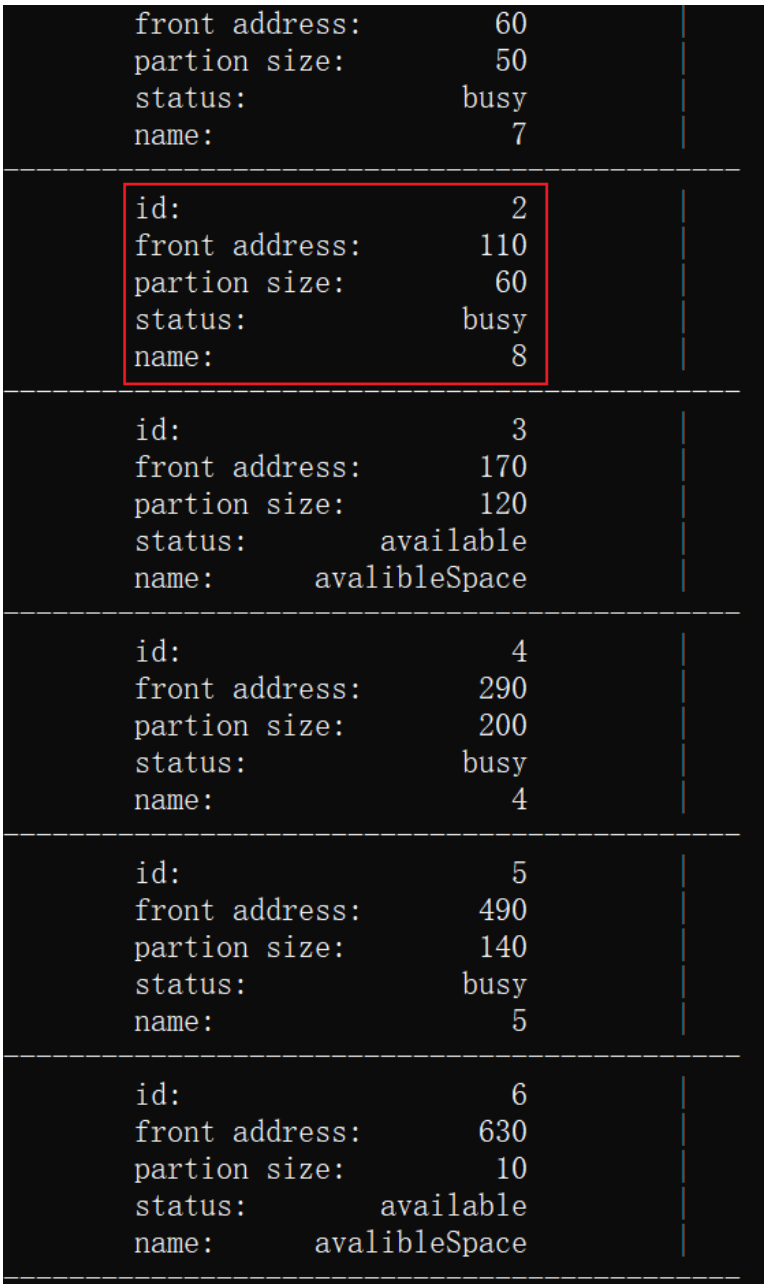
作业 7 申请 50KB



注：因信息过长，无法完全截图，因此保留关键的截图信息。

当前的两个空闲分区链中第一个分区大小为 10，不满足要求；第二个分区大小为 230，满足要求，于是对第二个空闲分区进行分配。分配情况如上图所示。剩余两个空闲分区，大小分别为 10，180.

作业 8 申请 60KB：



当前的两个空闲分区链中第一个分区大小为 10，不满足要求；第二个分区大小为 180，满足要求，于是对第二个空闲分区进行分配。分配情况如上图所示。剩余两个空闲分区，大小分别为 10，120。

七、总结

FF 每次分配总是从低址部分开始查找并分配，这样，低址部分因为不断地分割，可能会留下许多难以利用的碎片，降低了空间的利用率。而 BF 似乎比较好，但 BF 存在一些缺点：每次分配后所切割下来的剩余部分总是最小的，这样，就会在存储器中留下许多难以利用的碎片。

实验三　请求调页存储管理方式的模拟

一、实验目的

通过对页面、页表、地址转换和页面置换过程的模拟，加深对请求调页系统的原理和实现过程的理解。

二、实验内容

（1）假设每个页面中可存放 10 条指令，分配给作业的内存块数为 4。

（2）用 C 语言模拟一个作业的执行过程，该作业共有 320 条指令，即它的地址空间为 32 页，目前它的所有页都还未调入内存。在模拟过程中，如果所访问的指令已在内存，则显示其物理地址，并转下一条指令。如果所访问的指令还未装入内存，则发生缺页，此时需记录缺页的次数，并将相应页调入内存。如果 4 个内存块均已装入该作业，则需进行页面置换，最后显示其物理地址，并转下一条指令。在所有 320 指令执行完毕后，请计算并显示作业运行过程中发生的缺页率。

（3）置换算法：采用先进先出（FIFO）、最近最久未使用（LRU）和最佳置换（OPT）算法置换算法。

三、实现思路

首先需要随机生成 320 个数，数值范围限制在 0~319 之间，代表 320 条不同的指令，并将其存放在数组 temp 中；

然后计算每条指令对应的页面号是多少。每个页面可以存放 10 条指令，因此可以将随机生成的数值对 10 进行整除，商相同的表示处于同一个页面中。比如指令 174 和 176，分别对 10 整除，商都是 17，表示这两条指令处于页面号为 17 的页面中，在进行页面置换时，会一起被加载到内存中；

接着对内存块进行初始化。我们使用结构体 BLOCK（BLOCK 定义见下文）来对内存块进行映射。最开始没有调入页面，因此内存块是空的，即将其页号置为-1，访问次数 next_access_instruction 置为 0；

FIFO（先进先出）页面置换算法总是淘汰最先进入内存页面的，即选择在内存中驻留时间最久的页面予以淘汰。下面介绍使用 FIFO 来进行页面置换的过程。首先查找当前指令是否在内存块中，若在，则不需要进行页面置换，并且将除了

当前指令所在的内存块外的所有页面的 next_access_instruction 字段（表示未访问时间）加 1；若不在，则需要进行请求调页。请求调页分为两种情况：一种是内存块未满时，可直接将要调入的页面调入到内存块中；若内存块已满，则需要进行页面置换。进行页面置换的过程如下：比较内存块的 next_access_instruction 字段，找到第一个最大字段所在的内存块，将其置换下去，被要调入的新内存块更换。

LRU（最近最久未使用）算法的策略如下：选择最近最久未使用的页面予以淘汰。该算法赋予每个页面一个访问字段，用来记录一个页面自上次被访问以来所经历的时间 t。当需要淘汰一个页面时，选择现有页面中 t 值最大的，也就是最近最久未使用的页面予以淘汰。书本介绍要实现其过程需要硬件支持，本算法简化其过程，当内存中的某个物理块被访问时，就将其字段 next_access_instruction 置为 0。当选择页面进行淘汰时，选择字段 next_access_instruction 第一个最大值的页面进行淘汰，并将新换入的页号的字段 next_access_instruction 置为 0。

OPT（最佳置换）所选择的被淘汰页面将是以后永不使用的，或许是在最长时间内不再被访问的页面。那么如何确定哪一个页面是未来最长时间内不再被访问的呢？这个没有标准答案，且难以判断。为了简单起见，本算法采取的策略是判断最近要访问的指令中，其所在的页面号是否存在于内存块中，若存在，则将字段 next_access_instruction 置为该指令在数组 temp（temp 为随机指令数组）中的序号；若不存在，则将字符 next_access_instruction 置为一个很大的数（本例中置为 1000）。在进行页面置换是，选择字段 next_access_instruction 第一个最大值的页面进行淘汰。

四、主要的数据结构

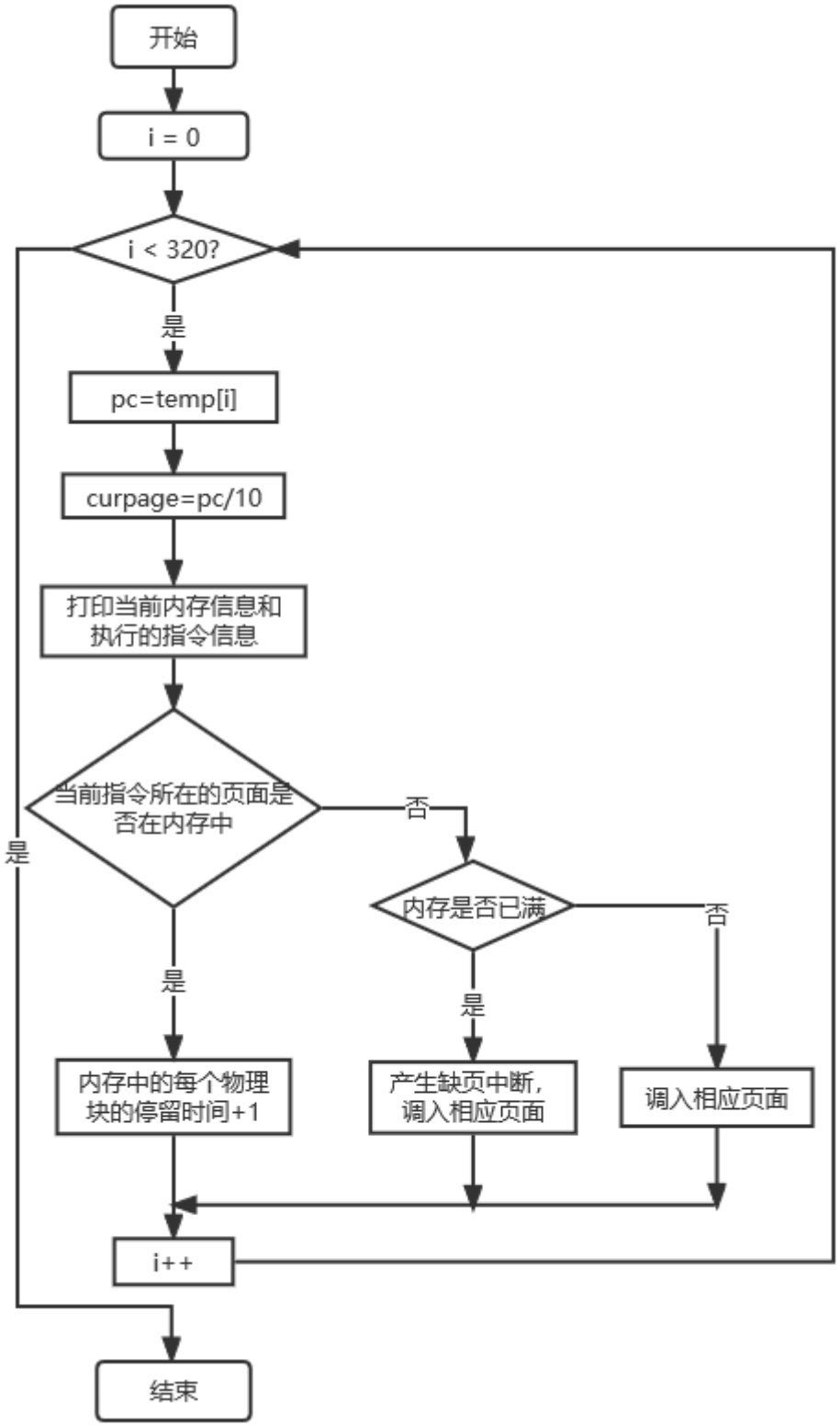
内存块的结构体声明

```
struct BLOCK
{
    int pagenum ;     // 页号
    int next_access_instruction;     // 未访问时间
};
```

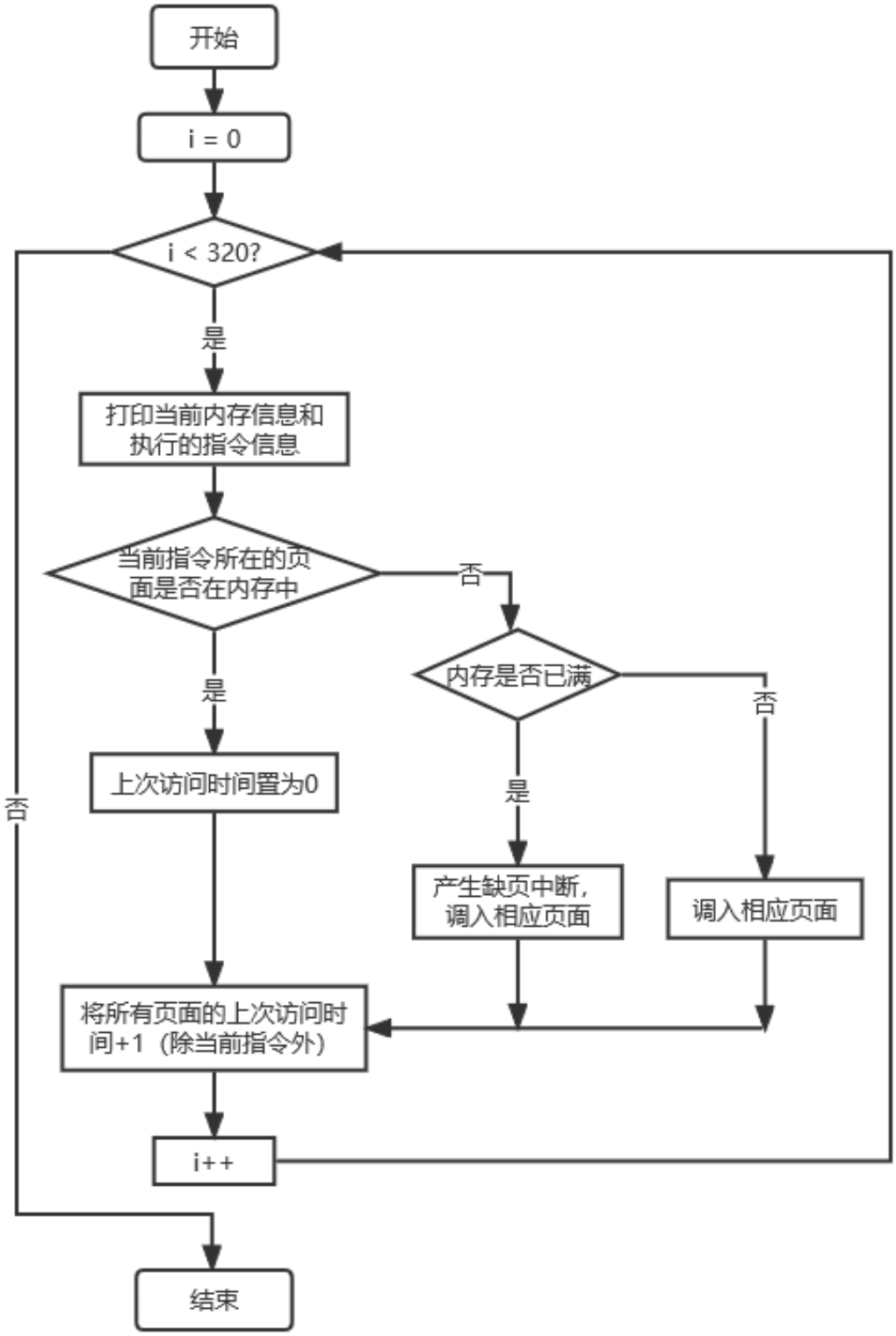
五、算法流程图

下面展示三种方法的程序流程图。

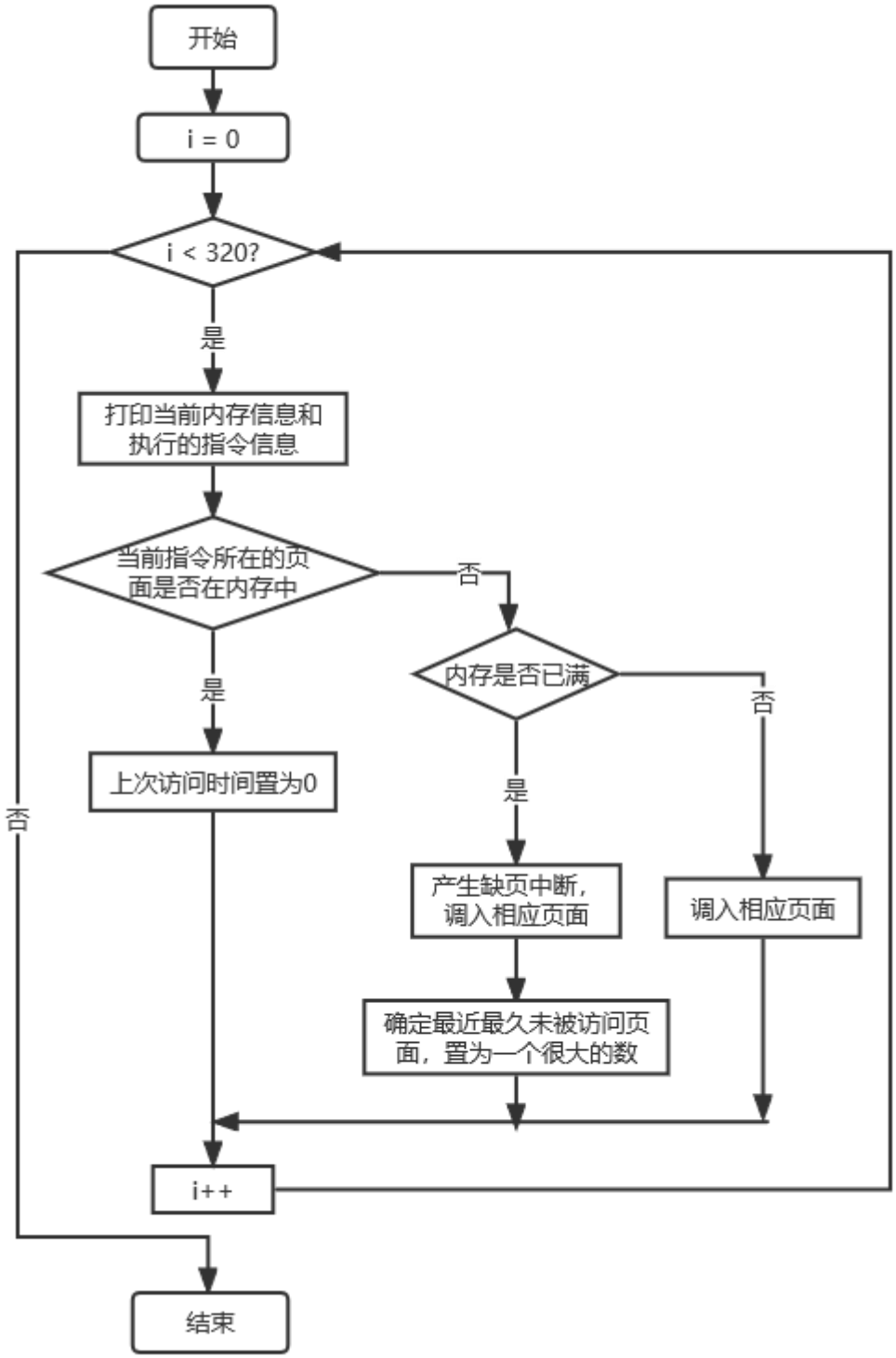
FIFO 流程图：


开始
i = 0
i < 320?
是
pc=temp[i]
curpage=pc/10
打印当前内存信息和
执行的指令信息
当前指令所在的页面是
否在内存中
否
是
内存是否已满
否
是
是
内存中的每个物理
块的停留时间+1
产生缺页中断，
调入相应页面
调入相应页面
i++
结束

LRU 流程图：


开始
i = 0
i < 320?
是
打印当前内存信息和
执行的指令信息
当前指令所在的页
面是否在内存中
否
是
内存是否已满
否
是
上次访问时间置为0
产生缺页中断，
调入相应页面
调入相应页面
否
将所有页面的上次访问时
间+1（除当前指令外）
i++
结束

OPT 流程图：


开始
i = 0
i < 320?
是
否
打印当前内存信息和
执行的指令信息
当前指令所在的页
面是否在内存中
否
是
内存是否已满
是
否
上次访问时间置为0
产生缺页中断，
调入相应页面
调入相应页面
确定最近最久未被访问页
面，置为一个很大的数
i++
结束

六、运行与测试

FIFO 测试过程

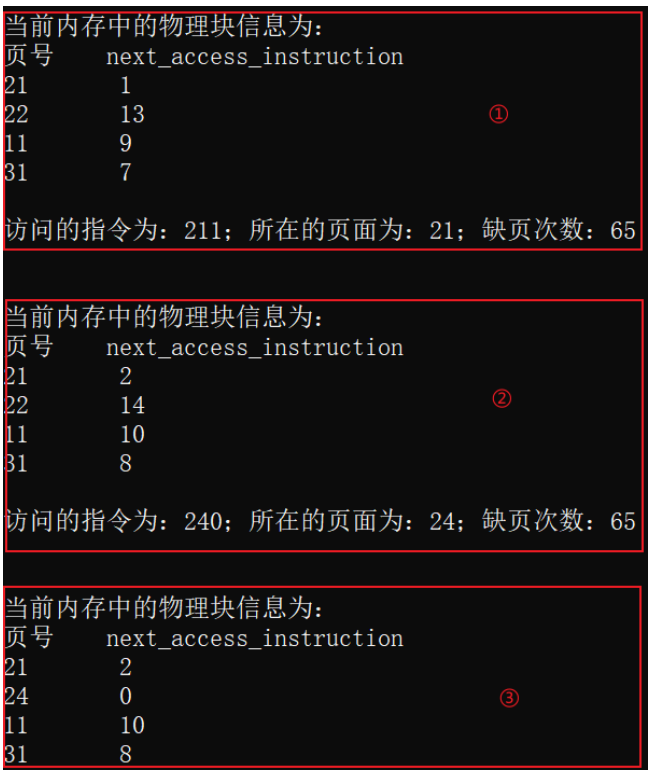
产生的随机数（部分）如下所示：

```
******按照要求产生的320个随机数：*******
 124 125 041 042 227 228 175 176 227 228
 138 139 209 210 114 115 229 230 086 087
 236 237 191 192 249 250 198 199 205 206
 198 199 276 277 040 041 096 097 071 072
 191 192 181 182 316 317 199 200 306 307
 012 013 153 154 087 088 115 116 054 055
 142 143 119 120 306 307 006 007 234 235
```

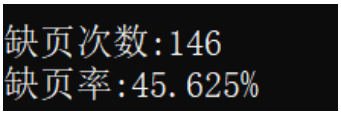
相应的页面号（部分）如下所示：

```
*****随机数对应的页面号*******
 12 12 04 04 22 22 17 17 22 22
 13 13 20 21 11 11 22 23 08 08
 23 23 19 19 24 25 19 19 20 20
 19 19 27 27 04 04 09 09 07 07
 19 19 18 18 31 31 19 20 30 30
 01 01 15 15 08 08 11 11 05 05
 14 14 11 12 30 30 00 00 23 23
```

下面选取算法执行过程中具有代表性的部分来讲解。在①中，当前访问的指令为 211，所在的页号为 21，内存中页号为 21 的页面已经调入内存块中，此时将字段 next_access_instruction（这里表示在内存中停留的时间）+1；在②中，当前访问的指令为 240，所在的页号为 24，并不在内存中，此时发生缺页中断，根据 FIFO 算法的策略，内存已满时，选取在内存中停留时间最长的，将其换出，在本例中，停留时间最长的为页号 22，将页号 22 换出内存，并调入页号 24，结果如③框内所示。



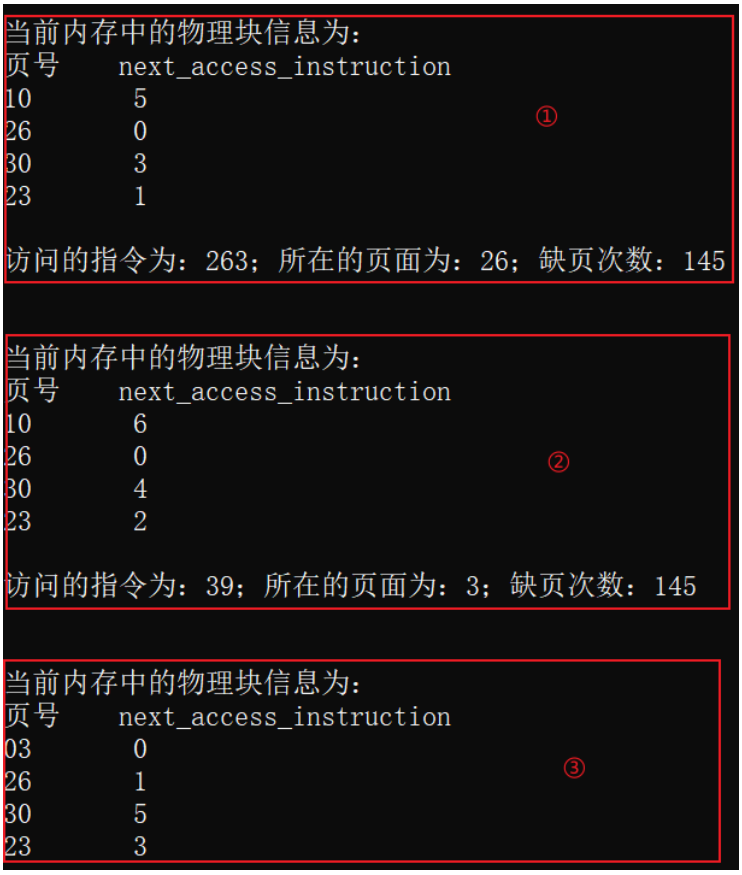
其余的以此类推。执行完 320 条指令后，缺页次数和缺页率如下所示：



（注：因指令是随机生成的，故缺页次数和缺页率在每次的结果不尽相同）

LRU 测试过程

同样，也是选取执行过程中最能体现出 LRU 算法策略的执行信息。在①框中，当前访问指令为 263，所在页面为 26，在内存中已经存在页号 26 了，于是将页号 26 的字段 next_access_instruction（此时表示自上次访问到现在的时间）置为 0，表示最近访问，而其它物理块的字段 next_access_instruction 均需要+1，处理完后的信息如②所示。在②框中，当前访问的指令为 39，所在的页面为 3，不在内存中，此时产生缺页中断，将字段 next_access_instruction 最大的换出，调入页面 3，结果如③框所示。



其余的以此类推。执行完 320 条指令后，缺页次数和缺页率如下所示：

OPT 测试过程

本测试过程也是选取最具代表性的执行过程进行解说。

```
当前内存中的物理块信息为:
页号     next_access_instruction
12        0
17        1000
13        313
14        1000

访问的指令为：130；所在的页面为：13；缺页次数：108


当前内存中的物理块信息为:
页号     next_access_instruction
12        0
17        1000
13        0
14        1000

访问的指令为：102；所在的页面为：10；缺页次数：108


当前内存中的物理块信息为:
页号     next_access_instruction
10        1000
17        1000
13        1000
```

其余的以此类推。执行完 320 条指令后，缺页次数和缺页率如下所示：

```
缺页次数:111
缺页率:34.6875%
```

七、总结

上面的测试过程在不同的条件下进行，即在不同的指令序列下进行。下表显示了在同一指令序列下的不同方法的缺页情况。

表 1 三种方法的比较

方法 / 指标	FIFO	LRU	OPT
缺页次数	146	144	111
缺页率	45.625%	45%	34.6875%

由表 1 可知，OPT 算法的效果最好，而 FIFO 和 LRU 算法的效果比较接近。

拓展实验

一、实验目的

分析操作系统的核心功能模块，理解相关功能模块实现的数据结构和算法，并加以实现，加深对操作系统原理和实现过程的理解。

二、实验内容

三、实现思路

四、主要的数据结构

五、算法流程图

六、运行与测试

七、总结